



Advanced Visual Effects with Direct3D®



Presenters: Cem Cebenoyan, Sim Dietrich, Richard Huddy,
Greg James, Jason Mitchell, Ashu Rege, Guennadi Riguer, Alex
Vlachos and Matthias Wloka



Today's Agenda

- **DirectX® 9 Features**
 - Jason Mitchell & Cem Cebenoyan

Coffee break – 11:00 – 11:15

- **DirectX 9 Shader Models**
 - Sim Dietrich & Jason L. Mitchell

Lunch break – 12:30 – 2:00

- **D3DX Effects & High-Level Shading Language**
 - Guennadi Riguer & Ashu Rege
- **Optimization for DirectX 9 Graphics**
 - Matthias Wloka & Richard Huddy

Coffee break – 4:00 – 4:15

- **Special Effects**
 - Alex Vlachos & Greg James
- **Conclusion and Call to Action**



DirectX® 9 Features



Jason Mitchell
JasonM@ati.com



Cem Cebenoyan
CCebenoyan@nvidia.com



Outline

- **Feeding Geometry to the GPU**
 - Vertex stream offset and VB indexing
 - Vertex declarations
 - Presampled displacement mapping
- **Pixel processing**
 - New surface formats
 - Multiple render targets
 - Depth bias with slope scale
 - Auto mipmap generation
 - Multisampling
 - Multihead
 - sRGB / gamma
 - Two-sided stencil
- **Miscellaneous**
 - Asynchronous notification / occlusion query



Feeding the GPU

In response to ISV requests, some key changes were made to DirectX 9:

- Addition of new stream component types
- Stream Offset
- Separation of Vertex Declarations from Vertex Shader Functions
- BaseVertexIndex change to DIP()



New stream component types

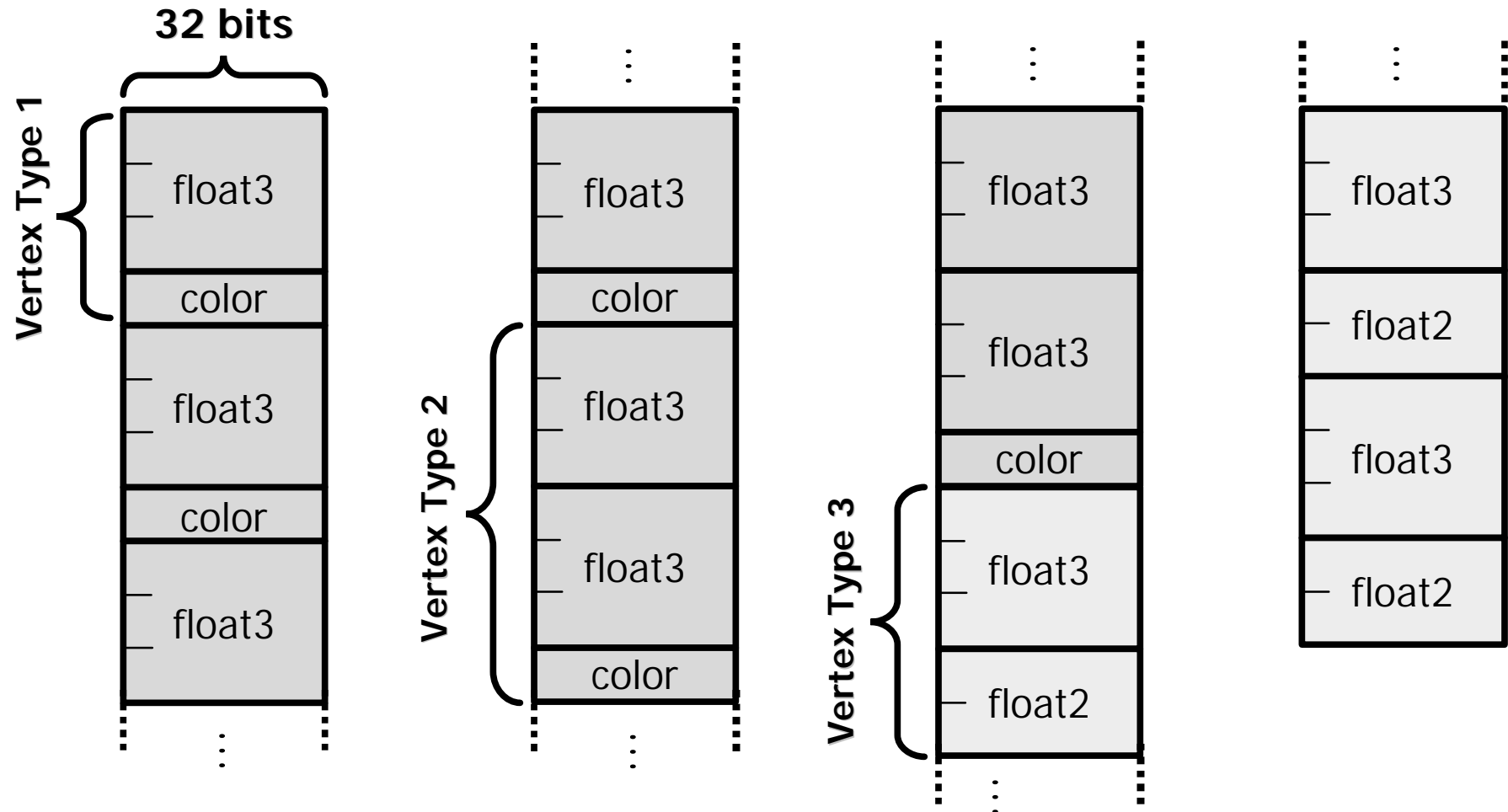
- `D3DDECLTYPE_UBYTE4N`
 - Each of 4 bytes is normalized by dividing by 255.0
- `D3DDECLTYPE_SHORT2N`
 - 2D signed short normalized ($v[0]/32767.0, v[1]/32767.0, 0, 1$)
- `D3DDECLTYPE_SHORT4N`
 - 4D signed short normalized ($v[0]/32767.0, v[1]/32767.0, v[2]/32767.0, v[3]/32767.0$)
- `D3DDECLTYPE_USHORT2N`
 - 2D unsigned short normalized ($v[0]/65535.0, v[1]/65535.0, 0, 1$)
- `D3DDECLTYPE_USHORT4N`
 - 4D unsigned short normalized ($v[0]/65535.0, v[1]/65535.0, v[2]/65535.0, v[3]/65535.0$)
- `D3DDECLTYPE_UDEC3`
 - 3D unsigned 10-10-10 expanded to (value, value, value, 1)
- `D3DDECLTYPE_DEC3N`
 - 3D signed 10-10-10 normalized & expanded to ($v[0]/511.0, v[1]/511.0, v[2]/511.0, 1$)
- `D3DDECLTYPE_FLOAT16_2`
 - Two 16-bit floating point values, expanded to (value, value, 0, 1)
- `D3DDECLTYPE_FLOAT16_4`
 - Four 16-bit floating point values



Vertex Stream Offset

- New offset in *bytes* specified in `SetStreamSource()`
- Easily allows you to place multiple objects in a single Vertex Buffer
 - Objects can even have different structures/strides
- New DirectX 9 driver is required
 - DirectX 9 drivers must set `D3DDEVCAPS2_STREAMOFFSET`
- Doesn't work with post-transformed vertices
- This isn't an excuse for you to go and make one big VB that contains your whole world

Vertex Stream Offset Example





Vertex Declarations

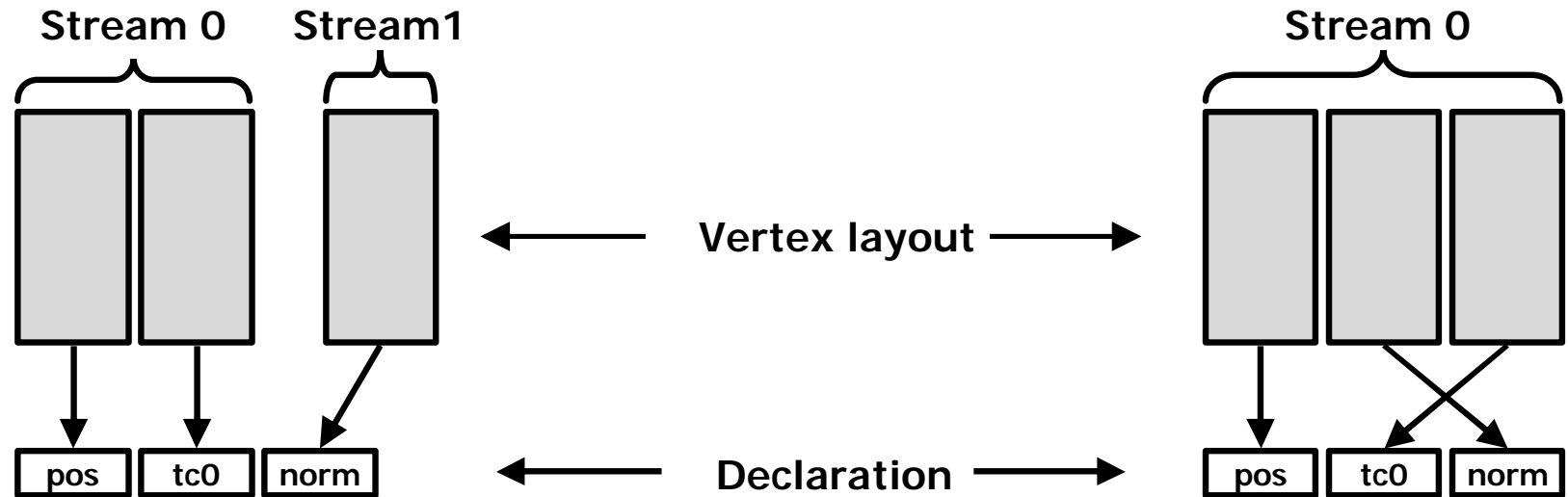
- The mapping of vertex stream components to vertex shader inputs is much more convenient and flexible in DirectX 9
- New concept of *Vertex Declaration* which is separate from the *Function*
- Declaration controls mapping of stream data to semantics
- Function maps from semantics to shader inputs and contains the code
- Declaration and Function are separate, independent states
- Driver matches them up at draw time
 - This operation can fail if function needs data the declaration doesn't provide



Semantics

- Usual Stuff:
 - POSITION, BLENDWEIGHT, BLENDINDICES, NORMAL, PSIZE, TEXCOORD, COLOR, DEPTH and FOG
- Other ones you'll typically want for convenience:
 - TANGENT, BINORMAL
- Higher-Order Primitives and Displacement mapping:
 - TESSFACTOR and SAMPLE
- Already-transformed Position:
 - POSITIONT
- Typically use `TEXCOORDn` for other engine-specific things
- Acts as symbol table for run-time linking of stream data to shader or FF transform input

Vertex Declaration



asm:

```
vs 1.1
dcl_position    v0
dcl_normal      v1
dcl_texcoord0   v2
mov r0, v0
...
```

HLSL:

```
VS_OUTPUT main (
float4 vPosition : POSITION,
float3 vNormal    : NORMAL,
float2 vTC0       : TEXCOORD0)
{
    ...
}
```



Creating a Vertex Declaration

Pass an array of **D3DVERTEXELEMENT9** structures to **CreateVertexDeclaration()**:

```
struct D3DVERTEXELEMENT9
{
    Stream;        // id from setstream()
    Offset;         // offset# verts into stream
    Type;           // float vs byte, etc.
    Method;         // tessellator op
    Usage;          // default semantic(pos, etc)
    UsageIndex      // e.g. texcoord[#]
}
```

Example Vertex Declaration

Array of D3DVERTEXELEMENT9 structures:

Usage
Index

Type

Method

Usage

Index

D3DVERTEXELEMENT9 mydecl[] =
{

```
{ 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},  
{ 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0},  
{ 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},  
{ 1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 1},  
{ 1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 1},  
{ 1, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 1},  
{ 2, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 2},  
{ 2, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 2},  
{ 2, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 2},  
D3DDECL_END()
```

};

Stream Offset



Creating a Vertex Shader Declaration

- **Vertex Stream**
 - Pretty obvious
- **DWORD aligned Offset**
 - Hardware requires DWORD aligned - Runtime validates
- **Stream component Type**
 - As discussed earlier, there are some additional ones in DX9
- **Method**
 - Controls tessellator. Won't talk a lot about this today
- **Usage and Usage Index**
 - Think of these as a tuple:
 - Think of `D3DDECLUSAGE_POSITION, 0` as Pos_0
 - Think of `D3DDECLUSAGE_TEXCOORD, 2` as Tex_2
 - A given (Usage, Usage Index) tuple must be unique
 - e.g. there can't be two Pos_0 's
 - Driver uses this tuple to match w/ vertex shader func
- `D3DDECL_END()` terminates declaration



Matching Decls to Funcs

- New `dc1` instructions
 - These go at the top of the code of *all* shaders in DX9, *even vs.1.1*
 - These match the (Usage, Usage Index) tuples in the vertex declaration
 - Every `dc1` in the vertex shader func must have a (Usage, Usage Index) tuple in the current vertex declaration or DrawPrim will fail
 - HLSL compiler generates `dc1` instructions in bytecode based upon vertex shader input variables
- `dc1`s are followed by shader code
- More on this in shader section later...



SetFVF ()

- `SetVertexShaderDeclaration()` and `SetFVF()` step on each other
- Think of `SetFVF()` as shorthand for `SetVertexShaderDeclaration()` if you have a single stream that happens to follow FVF rules

DrawIndexedPrimitive

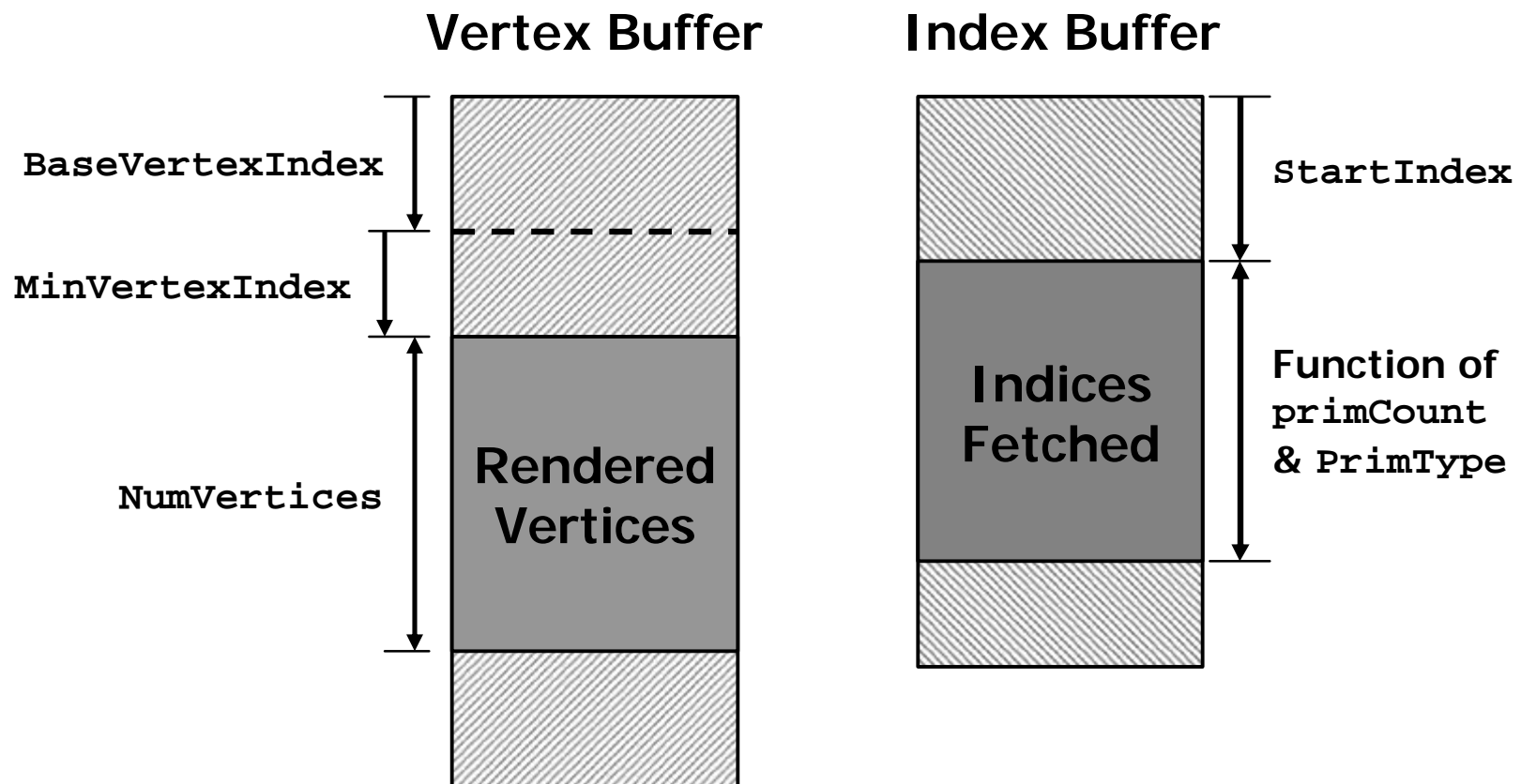
HRESULT

```
IDirect3DDevice9::DrawIndexedPrimitive(  
    D3DPRIMITIVETYPE PrimType,  
    INT BaseVertexIndex,  
    UINT MinVertexIndex,  
    UINT NumVertices,  
    UINT startIndex,  
    UINT primCount );
```

```
HRESULT IDirect3DDevice9::SetIndices(  
INT BaseVertexIndex,  
    IDirect3DIndexBuffer9* pIndexData );
```

- Does not require a DirectX 9 driver

Vertex Buffer Indexing





Higher Order Primitives

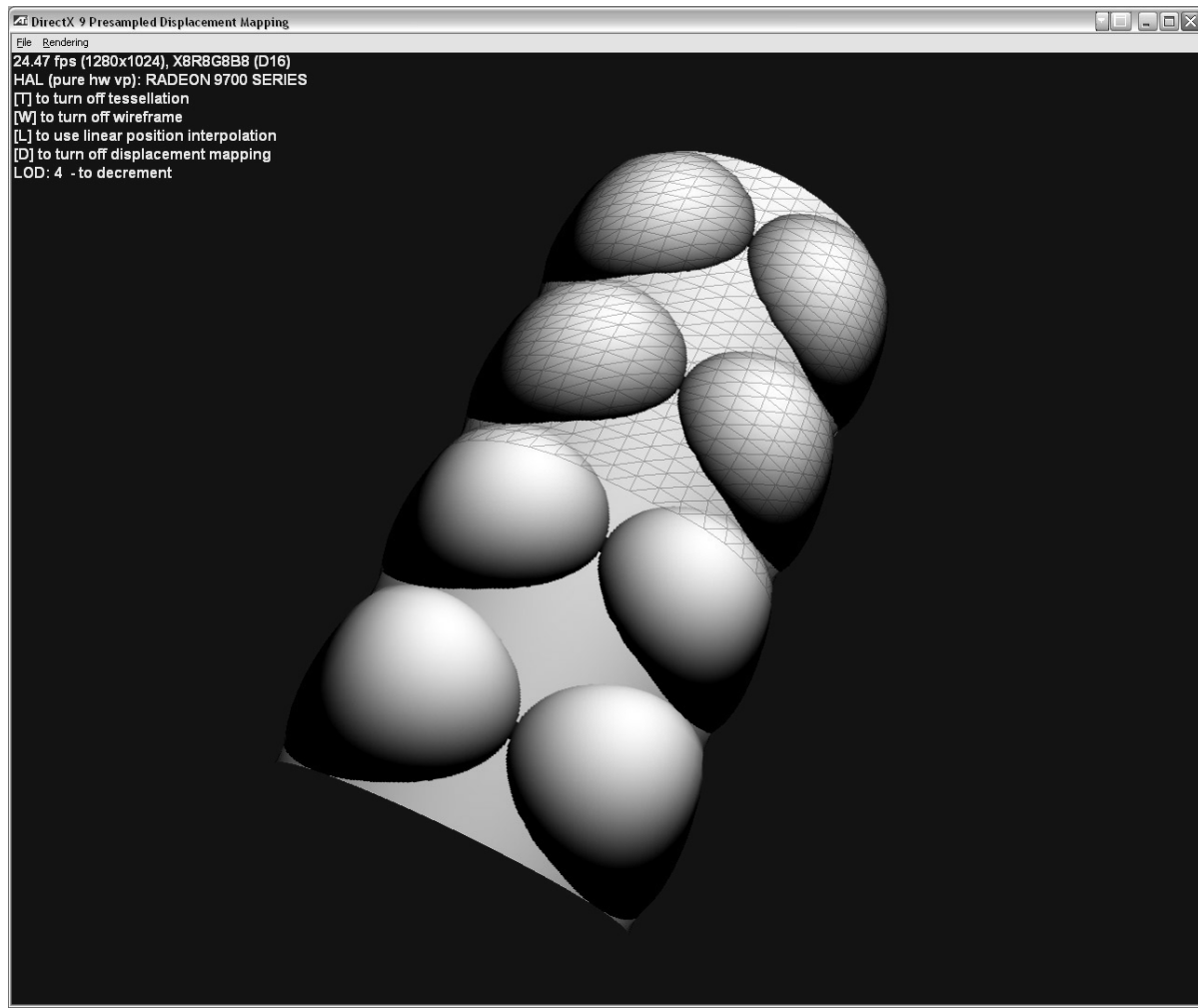
- **N-Patches have explicit call to enable and set tessellation level**
 - `SetNPatchMode(float* nSegments)`
- **Argument is number of segments per edge of each triangle**
- **Replaces previous renderstate**
- **Still captured in stateblocks**



Displacement Mapping

- Technique to add geometric detail by displacing vertices off of a mesh of triangles or higher order primitives
- Fits well with application LOD techniques
- But is it an API feature or an application technique?
- If the vertex shader can access memory, does displacement mapping just fall out?

Displacement Mapping



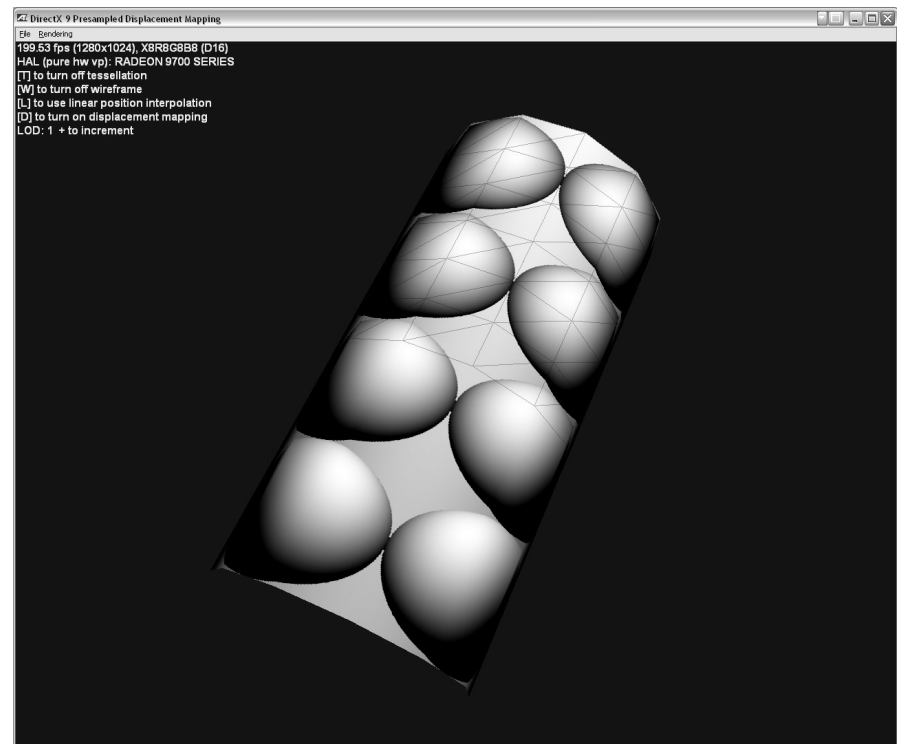
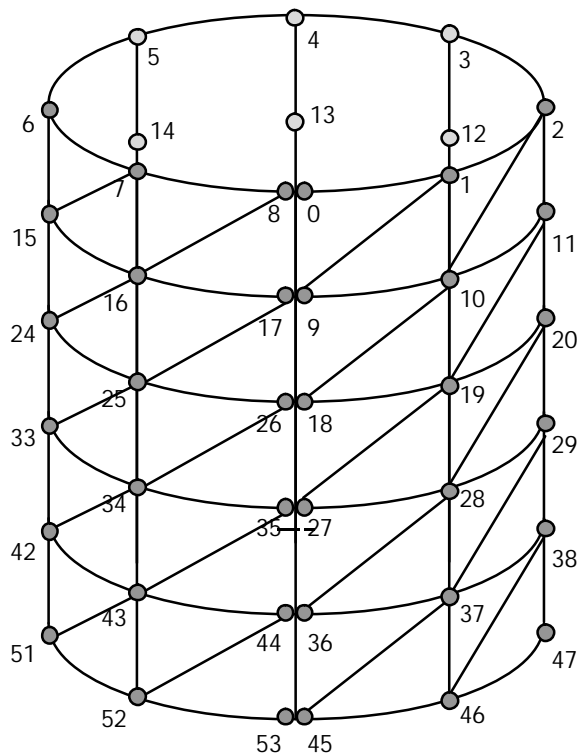
LOD1 Mesh



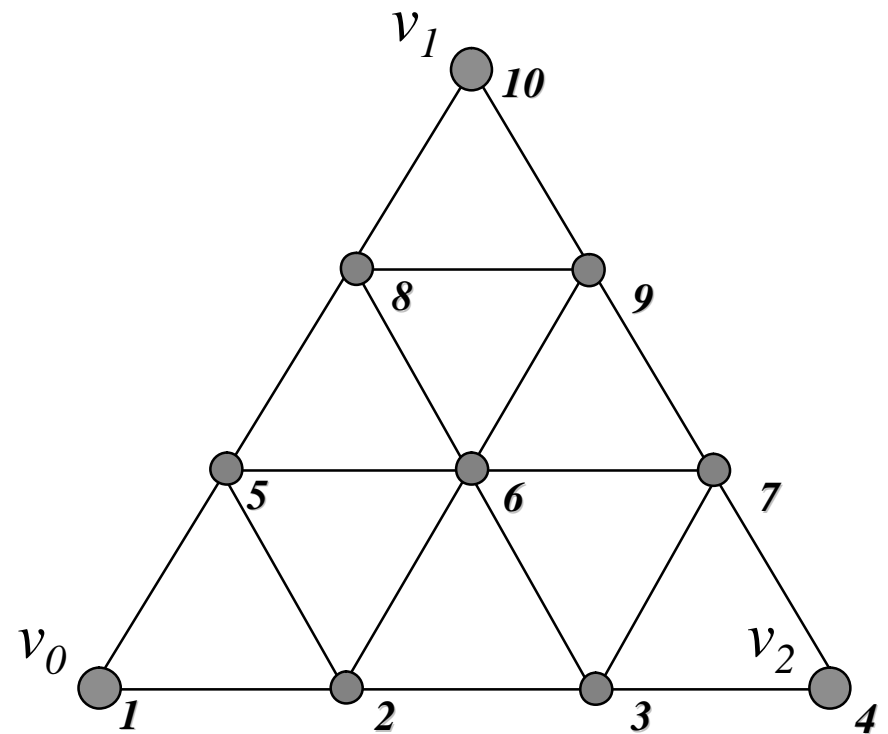
The coming unification...

- As many of you have asked us: *What's the difference between a surface and a vertex buffer anyway?*
- As we'll glimpse in the next section, the 3.0 vertex shader model allows a fairly general fetch from memory
- Once you can access memory in the vertex shader, you can do displacement mapping
- There is a form of this in the API today:
Presampled Displacement Mapping

Simple example



- Provide displacement values in a “linearized” texture map which is accessed by the vertex shader





Begin Cem



New Surface Formats

- Higher precision surface formats
 - D3DFMT_ABGR8
 - D3DFMT_ABGR10
 - D3DFMT_ABGR16
 - D3DFMT_ABGR16f
 - D3DFMT_ABGR32f
- Order is consistent with shader masks
- Note: ABGR16f format is s10e5 and has max range of approx +/-32768.0



Typical Surface Capabilities (March 2003)

• Format	Filter	Blend
• AGBR8	✓	✓
• ABGR10	✓	✓
• ABGR16	✓	x
• ABGR16f	x	x
• ABGR32f	x	x
• Use <code>CheckDeviceFormat()</code> with		
– <code>D3DUSAGE_FILTER</code> and <code>D3DUSAGE_ALPHABLEND</code>		



Higher Precision Surfaces

- **Some potential uses**
 - Deferred shading
 - FB post-processing
 - HDR
 - Shadow maps
 - Can do percentage closer filtering in the pixel shader
 - Multiple samples / larger filter kernel for softened edges



Higher Precision Surfaces

- However, current hardware has these drawbacks:
 - Potentially slow performance, due to large memory bandwidth requirements
 - Potential lack of orthogonality with texture types
 - No blending
 - No filtering
- Use `CheckDeviceFormat()` with
 - `D3DUSAGE_FILTER` and `D3DUSAGE_ALPHABLEND`



Multiple Render Targets

- Step towards rationalizing textures and vertex buffers
- Allow writing out multiple values from a single pixel shader pass
 - Up to 4 color elements plus Z/depth
 - Facilitates multipass algorithms



Multiple Render Targets

- **These limitations are harsh:**
 - **No support for FB pixel ops:**
 - Channel mask, a-blend, a-test, fog, ROP, dither
 - Only z-buffer and stencil ops will work
 - **No mipmapping, AA, or filtering**
 - **No surface Lock()**
- **Most of these will work better in the next hardware generation**



SetRenderTarget () Split

- Changed to work with MRTs
- Can only be one current ZStencil target
- RenderTargetIndex refers to MRT
- `IDirect3DDevice9::SetRenderTarget (`
 `DWORD RenderTargetIndex,`
 `IDirect3DSurface9* pRenderTarget);`
- `IDirect3DDevice9::SetDepthStencilSur`
 `face (IDirect3DSurface9*`
 `pNewZStencil);`



Depth Bias

- **Bias = m * D3DRS_ZSLOPESCALE + D3DRS_ZBIAS**
 - where, m is the max depth slope of triangle
 $m = \max(\text{abs}(\Delta z / \Delta x), \text{abs}(\Delta z / \Delta y))$
- **Cap Flag**
 - D3DPRASTERCAPS_SLOPESCALEDEPTHBIAS
- **Renderstates**
 - D3DRS_DEPTHBIAS, <float>
 - D3DRS_SLOPESCALEDEPTHBIAS, <float> -new
- **Important for depth based shadow buffers and overlaid geometry like tire marks**



Automatic Mip-map Generation

- **Very useful for render-to-texture effects**
 - Dynamic environment maps
 - Dynamic bump maps for water, etc.
- **Leverages hardware filtering**
 - That means it's fast, and done in whatever path the driver decides is optimal for this piece of hardware
- **Most modern GPUs can support this feature**



Automatic Mip-map Generation

- **Checking Caps**
 - `D3DCAPS2_CANAUTOGENMIPMAP`
- **Mipmaps can be auto-generated by hardware for any texture format (with the exception of DXTC compressed textures)**
- **Use `D3DUSAGE_AUTOGENMIPMAP` when creating the texture**
- **Filter Type**
 - `SetAutoGenFilterType(D3DTEXF_LINEAR);`
- **Mip-maps will automatically be generated**
 - Can force using `GenerateMipSubLevels()`



Scissor Rect

- Just after pixel shader
- API:
 - `D3DDevice9::SetScissorRect(*pRect);`
 - `D3DDevice9::GetScissorRect(*pRect);`
 - `D3DRS_SCISSORRECTENABLE`
- CAP:
 - `D3DPRASTERCAPS_SCISSORTEST`



Multisample Buffers

- Now supports separate control of
- Number of samples/pixel:
 - D3DMULTISAMPLE_TYPE
 - indicates number of separately addressable subsamples accessed by mask bits
- Image quality level:
 - DWORD dwMultiSampleQuality
 - 0 is base/default quality level
 - Driver returns number of quality levels supported via CheckDeviceMultisample()



Multihead

- All heads in a multihead card can be driven by one Direct3D device
 - So video memory can be shared
- Fullscreen only
- Enables dual and triple head displays to use same textures on all 3 display devices

Multihead

- New members in D3DCAPS9
 - `NumberOfAdaptersInGroup`
 - `MasterAdapterOrdinal`
 - `AdapterOrdinalInGroup`
- One is the Master head and other heads on the same card are Slave heads
- The master and its slaves from one multi-head adapter are called a Group
- `CreateDevice` takes a flag (`D3DCREATE_ADAPTERGROUP_DEVICE`) indicating that the application wishes this device to drive all the heads that this master adapter owns



Multihead Examples

Wacky Example

	Single-head card	Dual-head card		Triple-head card		
Adapter Ordinal	0	1	2	3	4	5
NumberOfAdaptersInGroup	1	2	0	3	0	0
MasterAdapterOrdinal	0	1	1	3	3	3
AdapterOrdinalInGroup	0	0	1	0	1	2

Real Example

	Dual-head card	
Adapter Ordinal	0	1
NumberOfAdaptersInGroup	2	0
MasterAdapterOrdinal	0	0
AdapterOrdinalInGroup	0	1



Constant Blend Color

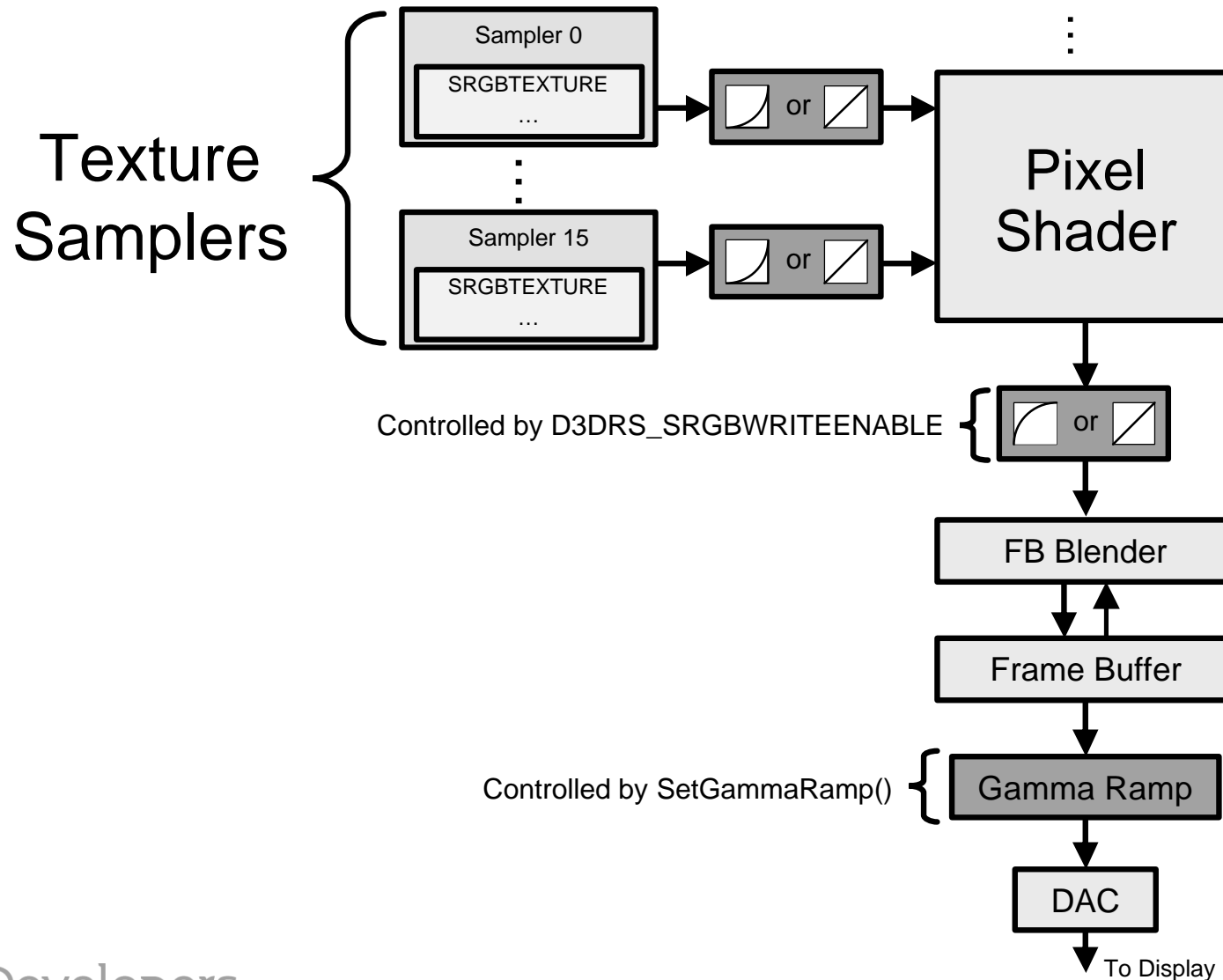
- An additional constant is now available for use in the frame-buffer blender
- This is supported in most current hardware
- Set using **D3DRS_BLENDFACTOR** dword packed color
- Use in blending via
 - **D3DBLEND_BLENDFACTOR**
 - **D3DBLEND_INVBLENDFACTOR**



sRGB

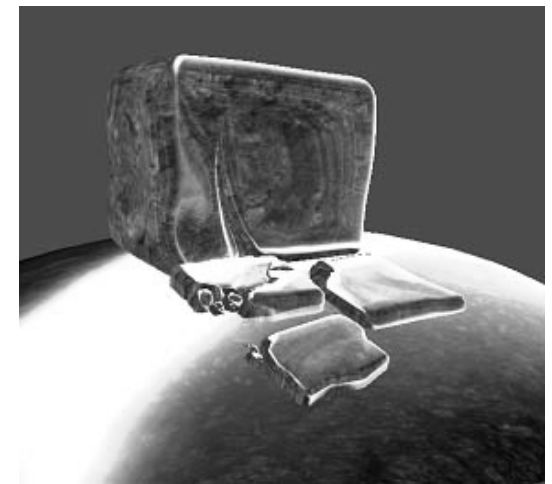
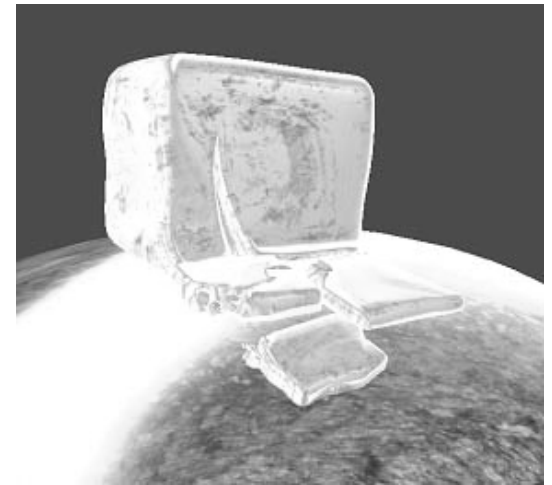
- Microsoft-pushed industry standard (g 2.2) format
- In Direct3D, sRGB is a sampler state, not a texture format
- May not be valid on all texture formats, however
 - Determine this through `CheckDeviceFormat` API

sRGB and Gamma in DirectX 9



sRGB

- Symptoms of ignoring gamma:
- Screen/textures may look washed out
 - Low contrast, greyish
- Addition may seem too bright
- Division may seem too dark
 - $\frac{1}{2}$ should be 0.73
- User shouldn't have to adjust monitor





sRGB

- **Problem**

- Math in gamma space is not linear ($50\% + 50\% \neq 1.0$)
 - Input textures authored in sRGB
- Math in pixel shader is linear ($50\% + 50\% = 1.0$)

- **Solution**

- Texture inputs converted to linear space (rgb?)
 - D3DUSAGE_QUERY_SRGBREAD
 - D3DSAMP_SRGBTEXTURE
- Pixel shader output converted to gamma space (rgb1/?)
 - D3DUSAGE_QUERY_SRGBWRITE
 - D3DRS_SRGBWRITEENABLE
 - Limited to the first element of MET



sRGB

- sRGB defined only for 8-bit unsigned RGB surfaces
 - Alpha is linear
- Color clears are linear
- Windowed applications either
 - Perform a gamma correction blit
 - Or use D3DPRESENT_LINEAR_CONTENT if exposed
 - D3DCAPS3_LINEAR_TO_SRGB_PRESENTATION
- Frame buffer blending is *NOT* correct
 - Neither is texture filtering
- D3DX provides conversion functionality



Two-sided Stencil

- Stencil shadows volumes can now be rendered in 1 pass instead of two
 - Biggest savings is in transform
- Check caps bit
 - D3DSTENCILCAPS_TWOSIDED
- Set new render state to TRUE
 - D3DRS_TWOSIDEDSTENCILMODE
- Current stencil ops then apply to CW polygons
- A new set then applies to CCW polygons
 - D3DRS_CCW_STENCILFAIL
 - D3DRS_CCW_STENCILPASS
 - D3DRS_CCW_STENCILFUNC



Discardable Depth-Stencil

- Significant performance boost on some implementations
- Not the default: App has to ask for discardable surface in presentation parameters on Create or it will not happen
- If enabled, implementation need not persist Depth/Stencil across frames
- Most applications should be able to enable this



Asynchronous Notification

- Mechanism to return data to app from hardware
- App posts query and then can poll later for result without blocking
- Works on some current and most future hardware
- Most powerful current notification is “occlusion query”



Occlusion Query

- Returns the number of pixels that survive to the framebuffer
 - So, they pass the z test, stencil test, scissor, etc.
- Useful for a number of algorithms
 - Occlusion culling
 - Lens-flare / halo occlusion determination
 - Order-independent transparency



Occlusion Query – Example

- Create `IDirect3DQuery9` object
 - `CreateQuery(D3DQUERYTYPE_OCCLUSION)`
 - You can have multiple outstanding queries
- `Query->Issue(D3DISSUE_BEGIN)`
- Render geometry
- `Query->Issue(D3DISSUE_END)`
- Potentially later, `Query->GetData()` to retrieve number of rendered pixels between Begin and End
 - Will return `S_FALSE` if query result is not available yet



Occlusion Query – Light halos

- Render light's geometry while issuing occlusion query
- Depending on the number of pixels passing, fade out a halo around the light
- If occlusion info is not yet available, potentially just use the last frame's data
 - Doesn't need to be perfect



Occlusion Query - Multipass

- A simple form of occlusion culling
- If a rendering equation takes multiple passes, use occlusion queries around objects in the initial pass
- In subsequent passes, only render additional passes on objects where the query result $\neq 0$
 - Doesn't cost perf because occlusion query around geometry you're rendering anyway is "free"



Summary

- **Feeding Geometry to the GPU**
 - Vertex stream offset and VB indexing
 - Vertex declarations
 - Presampled displacement mapping
- **Pixel processing**
 - New surface formats
 - Multiple render targets
 - Depth bias with slope scale
 - Auto mipmap generation
 - Multisampling
 - Multihead
 - sRGB / gamma
 - Two-sided stencil
- **Miscellaneous**
 - Asynchronous notification / occlusion query



Coffee Break

**We will start back up
again at 11:15**